PPP: The Pan Program Presenter *

Christina L. Black

November 5, 1990

Abstract

Semantically-powerful views, formatted program text, and structural operations on programs are powerful tools for helping users understand and edit programs. The Pan Program Presenter provides these facilities, including elision and display of program annotations, in an incremental, portable implementation. It provides novel languages for specifying program appearance, including both a succinct, easy-to-use form and a more powerful, extensible form.

*This research was sponsored by the Defense Advance Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292. Christina L. Black was supported in part by a National Science Foundation Graduate Fellowship, by a grant from the Bell Laboratories Graduate Research Program for Women, by a gift from Joseph D. and Carolyn E. Tajnai, and by a matching gift from Hewlett-Packard.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **05 NOV 1990**	2. REPORT TYPE	3. DATES COVERED **00-00-1990 to 00-00-1990**
4. TITLE AND SUBTITLE **PPP: The Pan Program Presenter**		5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S) 11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Semantically-powerful views, formatted program text, and structural operations on programs are powerful tools for helping users understand and edit programs. The Pan Program Presenter provides these facilities, including elision and display of program annotations, in an incremental, portable implementation. It provides novel languages for specifying program appearance, including both a succinct, easy-to-use form and a more powerful, extensible form.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**60**	

Contents

1 Introduction: Problems and Solutions

Pan I is a system designed to make programmers more effective by giving them powerful ways of viewing and editing programs [BV87] [BGV90]. Since so little is known about what are the most useful viewing and editing paradigms, *Pan I* is flexible, serving as a test bed for comparing and developing these paradigms. A program presentation tool has much to offer in pursuit of these goals of power and flexibility. It can provide increased access to structural ways of thinking about and manipulating programs; it can provide formats that make syntactic structure apparent and thus make programs easier to read; and it can provide semantically powerful viewing mechanisms. If it is sufficiently flexible, it can allow researchers to experiment with ways of using these three techniques to empower users.

For a tool providing these services to be useful, it must be incremental, since otherwise it will be too slow to use on large programs. It must also be flexible and description-driven, since we will want to display a given program differently based on user preferences or the kind of viewing or editing task being performed, experiment with different approaches to program presentation, and allow presentation of programs written in different languages. Finally, so that it can remain useful as *Pan I* and its successor system Ensemble evolve, it should be portable, so that it can be used to deal with non-textual representations, and to deal with implementations of text other than that currently used in *Pan I*.

The Pan Program Presenter (PPP) provides structural, formatted, and semantically-powerful views of programs in the *Pan I* context in an incremental, description-driven, and portable way, using novel specification paradigms.

1.1 Structural Operations

One of *Pan I*'s design goals has always been to support both textual and structural viewing and editing operations on programs. Structural operations are powerful, but users are accustomed to using textual operations. Because users have highly-developed skills for textual manipulation of programs, such manipulation is highly effective. Moreover, users are likely to feel hamstrung by a system that does not allow such manipulation, and will probably not use it. Nevertheless, *Pan I*'s developers believe that structural operations

are valuable and powerful, and that, if they are supplied along with textual operations, users will gradually adopt them, at least in the situations where they provide more power than textual manipulations.

However, *Pan I*'s access to structure has been limited in the past. While structural selection and navigation were possible, purely structural editing of programs was not possible. The reason lies in the implementation architecture. Every program is represented by an abstract syntax tree whose leaves contain pointers into the text of the program as edited by the user. The user can make additional textual edits to the text stream; the tree is updated to reflect these edits via incremental parsing. The textual and tree representations of programs have always had equal status. Neither has been given primacy, and there is no clear paradigm for structural manipulations. For example, it was impossible to do purely structural transformations, because there was no way to generate text for the newly-created structure. The program presenter gives us a tool that allows creation of textual views from the (now-primary) tree representation. Textual editing remains possible, since we can use a parser to transform textual edits into structural edits. Structural editing becomes possible, because we can now provide views onto the changed objects.

These new capabilities have yet to be fully exploited. Textual editing in PPP-created views is impossible until a new parser is installed, and structural editing has not yet been implemented. However, the program presenter supplies the basic building-block that has heretofore been missing: views created solely from the program tree.

1.2 Formatted Program Text

Formatted program text, which uses layout and fonts to reflect lexical and syntactic structure, is valuable in enhancing program readability. [OC90] PPP provides this facility, allowing the user to specify the appearance of each program construct. Fonts and whitespace can be used to reflect such structural properties as lexical class and nesting depth. See for example Figure 1 which shows unformatted and formatted views of the same program. Note that the formatted view's display panel says "language: Text," indicating that the view is purely textual, and does not allow structural navigation; no language-oriented commands are meaningful in this view. PPP does not maintain the mappings from formatted views to the to the tree which would

be needed to provide this facility; for discussion of possible extensions in this area, see Section 5.

1.3 Semantically-Powerful Views

Pure program text is not sufficiently informative. Sometimes it contains too much information; detail may obscure the overall structure of a program, or intervening, unrelated statements may make it impossible to view distantly-spaced but semantically-related statements simultaneously. In such cases, we may wish to *elide* (suppress display of) parts of the program. See for example Figure 2, in which error-handling code is elided to make it possible to view only the standard thread of control. Note that the two views use fonts differently; an elided view is not required to use the same formatting style as any other view. At other times, program code contains too little information. We may wish to display annotations on the program, or to influence the display of the program on the basis of computed values. When a view of a program reflects non-syntactic information about a program, whether that information is static semantics, computed information, such as dataflow information, or user annotations, we call the view *semantically-powerful.*

Pan I provides facilities for making computations about programs via *Colander*. *Colander* provides a language for specifying oontextual constraints (computations) on programs, an incremental constraint-checker which updates computations in response to user editing events, and a database of information about programs being edited which contains both information computed by *Colander* and information computed by other *Pan I* clients. PPP can display user comments and uses *Colander* to implement elision and indentation; this provides proof of concept of its ability to make use of annotations and semantic values.

1.4 Flexibility

As in much of *Pan I*, the design of PPP is oriented towards providing facilities that will allow later implementation of a wide range of policies, rather than implementing particular policies in the code. To this end, PPP is description-driven. Descriptions can be specified at three levels.

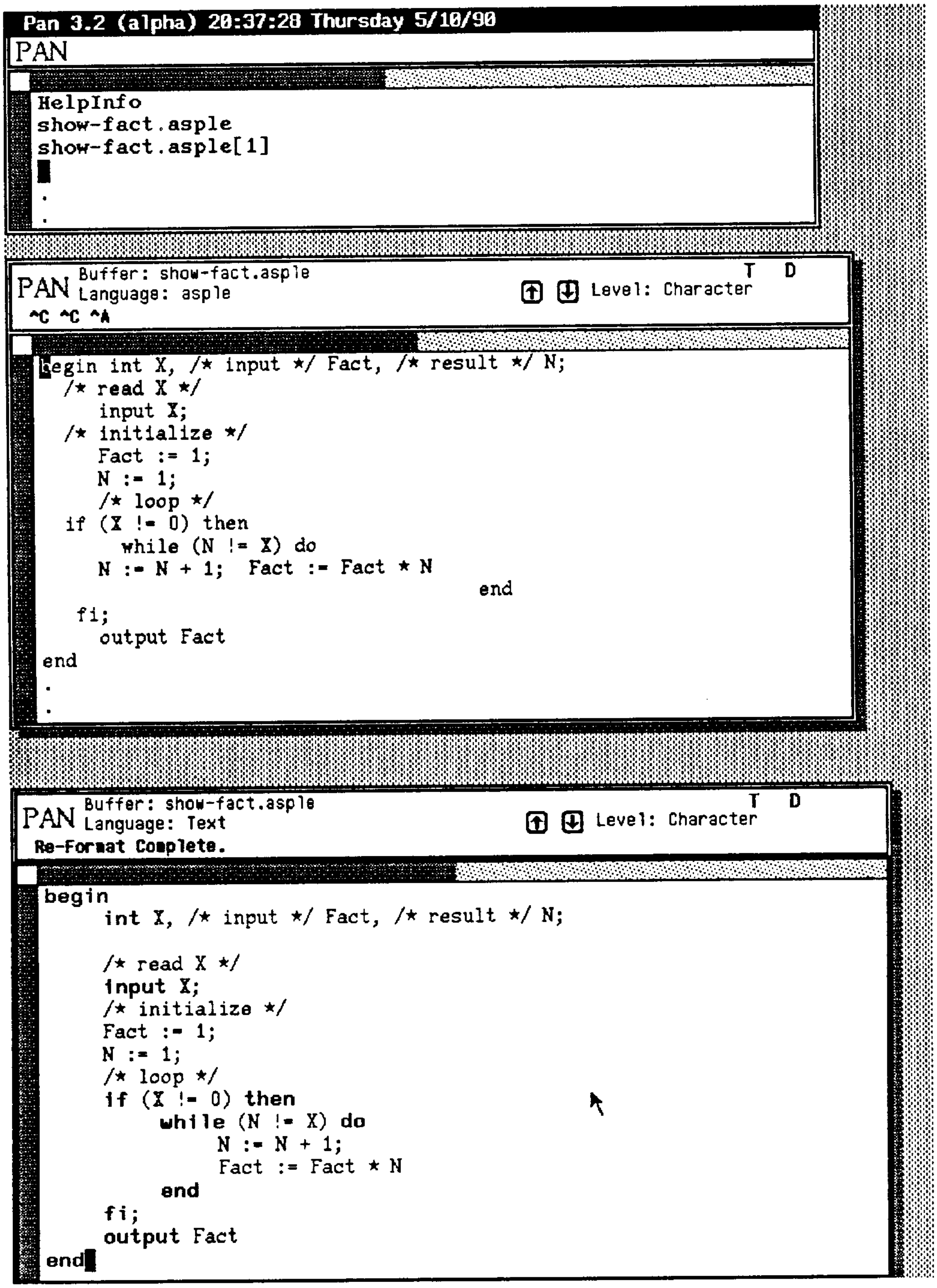

Figure 1: Formatted Program Text

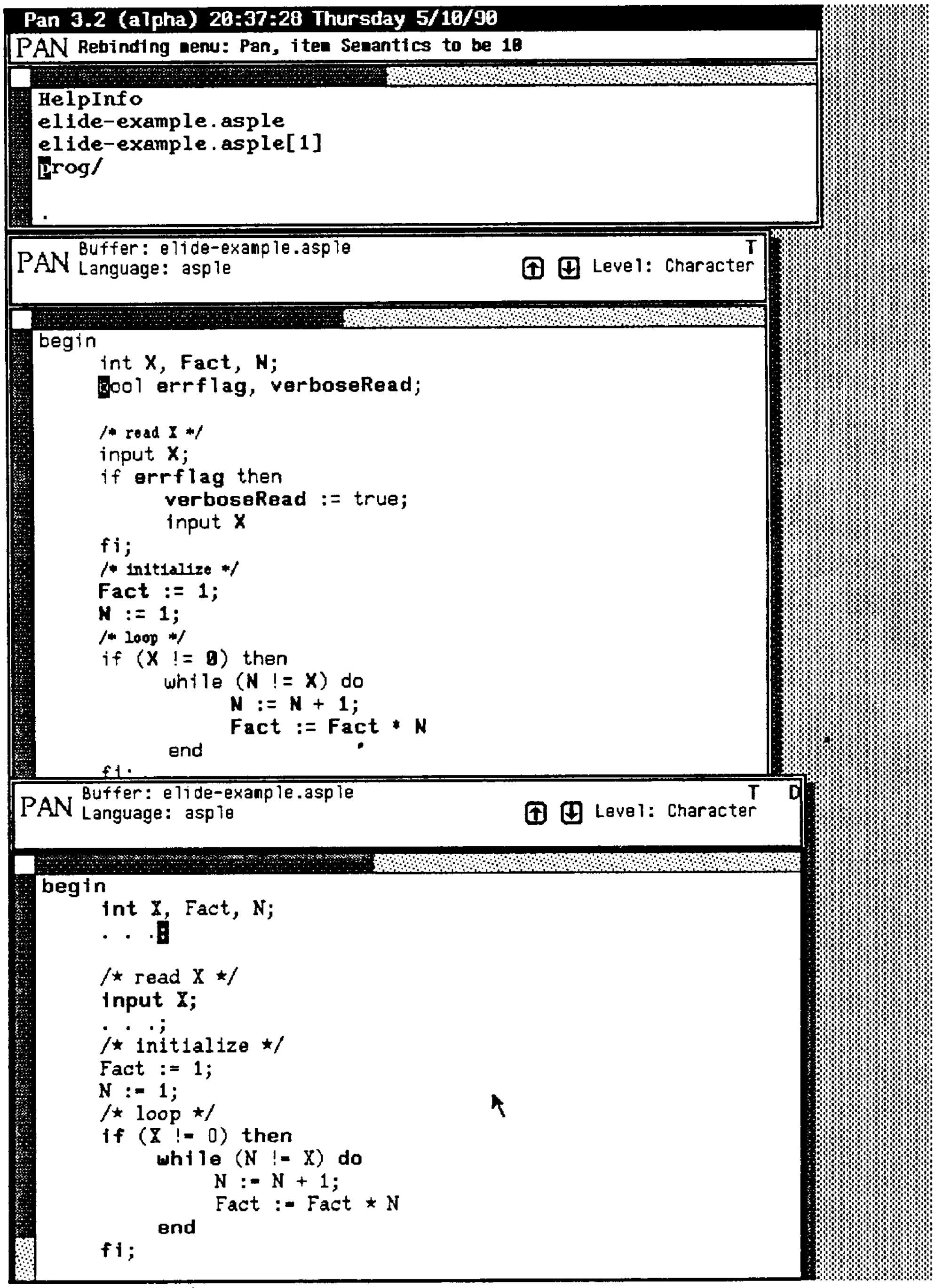

Figure 2: Elision

The status level gives a concise if highly-constrained way of specifying program appearance. It provides acceptable results in return for very little work in specification and is useful for providing presentations of language prototypes. Status-level descriptions can be processed to produce front end descriptions.

A complete program appearance specification consists of a front end specification and a back end specification. The front end level associates an appearance specification with each rule of the abstract grammar. These appearance specifications are expressed in terms of primitive appearance notions, such as character strings, spaces, and carriage returns. The back end level provides COMMON LISP code to implement each appearance primitive. This code can read, use, and display information from *Colander*'s database of computed values about the program.

A casual user can write a front end description to specify her formatting preferences and use it in conjunction with one of the back ends provided with the system. The standard back end will format her program on the screen; other back ends can provide other representations, such as document formatting languages. More advanced users can write back end specifications to provide semantically-powerful views of programs.

1.5 Incrementality

PPP is incremental; only the parts of the appearance that represent pieces of the tree that have changed are re-computed.

1.6 Portability

A PPP appearance specification is divided into two parts: a front end specification, which describes the desired appearance in terms of appearance primitives, and a back end specification, which describes how appearances are to be implemented. A given front end description can be used with any of several different back end descriptions, and can thus be used to produce the same appearance in different output representations. These description mechanisms will be described in more detail, with examples, in Section 3.

Currently, back ends are constrained to produce their output in the form of text rather than, for example, graphics. Even with this constraint, they are able to produce different representations. For example, one back end

produces text suitable for direct viewing; indentation is produced with space characters, and emphasized characters are printed in a bold font. Another back end produces TeX source, with indentation and emphasis represented by appropriate TeX commands. This text can be written to a file and processed by TeX to produce the desired appearance. This is useful, for example, in producing program examples to be included in textual documents. See Figure 3 for two views of a program, produced with the same front end description and different back end descriptions. One is suitable for direct viewing; the other is TeX source. The output produced by TeX from the latter version can be seen in Figure 4. With minor changes, the back end mechanism will provide true portability between representations.

1.7 System Context

Pan I [BV87] is a language-based editor written in COMMON LISP and C, running under SunView and X11. Its syntactic specifications are written in *Ladle* [But89]; its semantic specifications are written in *Colander* [Bal89]. Users who wish to view programs using PPP must be familiar with the *Pan I* user interface. Users who wish to write status and front end presentation descriptions for PPP must be familiar with *Ladle*, though only a surface familiarity is needed to write status descriptions. They must also be familiar with *Colander*. Users who wish to write back end descriptions should be familiar with COMMON LISP and the *Pan I* extension language.

1.8 Plan of Paper

Having addressed issues of design goals and motivations, this paper will proceed from the outside in. It begins in section 2 with a discussion of the way the system looks to a user who is simply using alternate views on her programs, using specifications provided with the system or by other users; it describes the capabilities the system provides and the user interface used to invoke them. This discussion includes coverage of elision. It proceeds, in Section 3, with a discussion of the various specification languages, from simplest to most complex, along with the tools the system provides for building and processing these descriptions. It finishes its inward journey in Section 4 with a description of the implementation. Sections 5 and 6 discuss further work to be done on PPP and list known bugs and problems. The final two sections

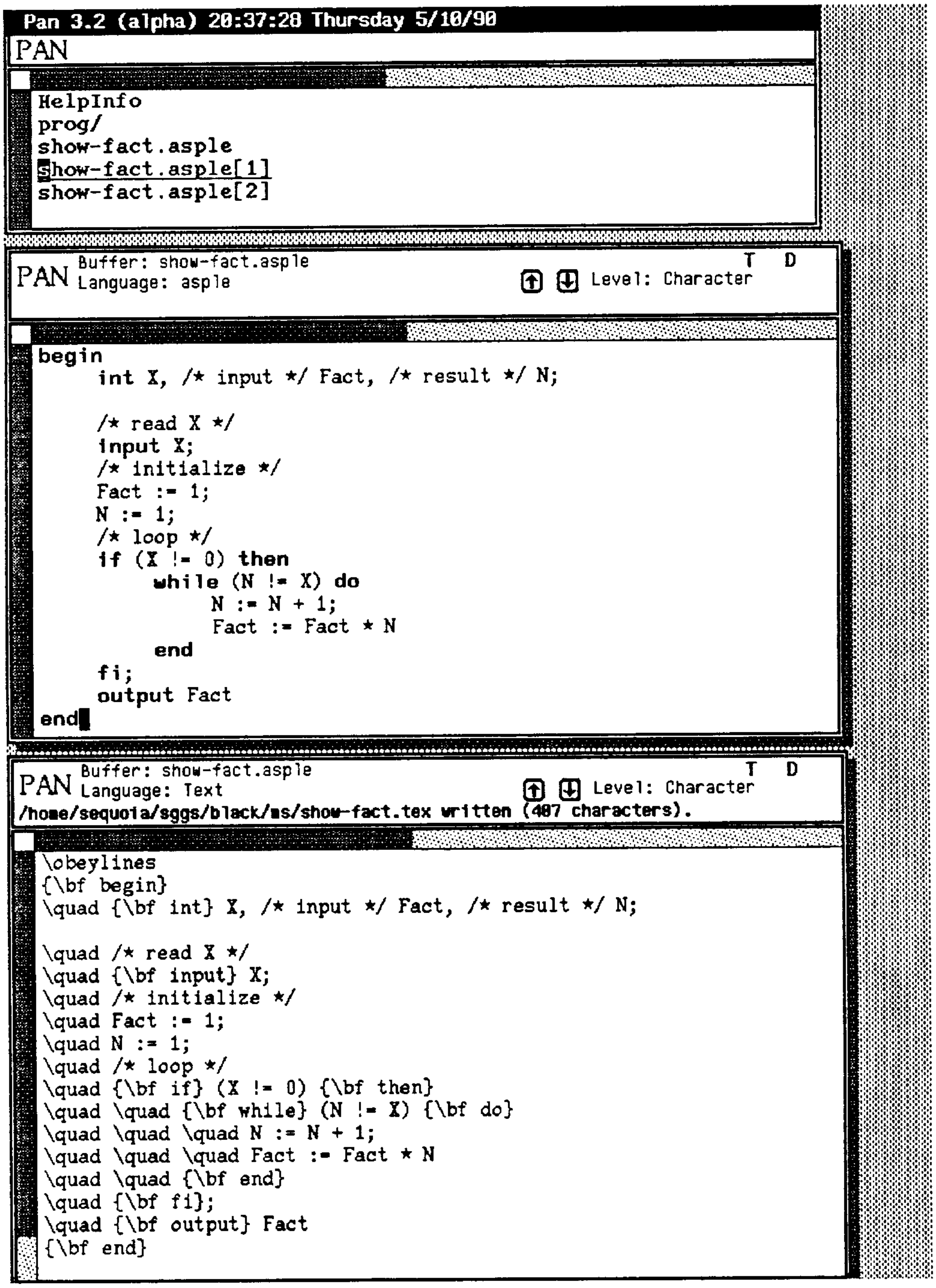

Figure 3: Two Different Back Ends

```
begin
  int X, /* input */ Fact, /* result */ N;
  /* read X */
  input X;
  /* initialize */
  Fact := 1;
  N := 1;
  /* loop */
  if (X != 0) then
    while (N != X) do
      N := N + 1;
      Fact := Fact * N
    end
  fi;
  output Fact
end
```

Figure 4: PPP Output Run Through TEX

put the project in perspective, with a discussion of PPP in the context of other work in the field in Section 7 and an overall evaluation of the project, listing its successes and failures, in Section 8. The appendices provide examples: a complete front end specification for **asple**, two back end descriptions, and a specification of **asple** in *Ladle*.

Throughout this paper, terms being defined are printed in *italics* and COMMON LISP objects and code are printed in `typewriter font`. *Pan I* commands and functions and macros in the *Pan I* extension language are printed in a sans-serif font.

1.9 Acknowledgments

Many people contributed to the success of this project. Thanks are due first of all to my advisor, Susan L. Graham, for technical and academic advice, not to mention extreme patience. Thanks to Michael A. Harrison for serving as second reader. Thanks also to all members, past and present, of the Piper and Ensemble research projects, who form the community within which I have worked. Especially important are my fellow Panpipers, who all

provided valuable technical advice and support: Bob Ballance, who started it all; Michael Van De Vanter, who keeps it from falling apart, and coordinated some pieces of system development that were on my critical path; Jacob Butcher, LISP hacker extraordinaire; Mark Hastings and Bruce Forstall, who kept me company during that long stretch of non-stop office occupancy, and helped make me feel like a grownup by being more junior than me; and Darrin Lane, who did our LISP port. Doug Grundman wrote the screen dump code which made many of the figures in this report possible. In addition to his work on Pan, Michael Van De Vanter provided valuable feedback on this report.

Not all help is technical and academic. Kathryn Crabtree helped me fight off the bureaucracy, listened to my moans, and loaned me mystery novels. Mark Sullivan and Margo Seltzer provided the kind of support which only a fellow-sufferer can give. And last but hardly least, Dana Bergen and Andy Wedel are the best friends I could hope to have, and kept me going and even smiling when things got rough.

2 The User Point of View

This section outlines the operations users can perform on programs using PPP. It is not a complete user manual, but rather aims to give the reader an overview of the services provided by PPP.

2.1 Visiting Programs and Creating Views

In *Pan I*, a user edits a program (or any other object) in a *buffer*. The buffer may contain multiple *views* on the object; currently, each view is textual. Each view may have zero or more *viewers*, or windows. The user initially imports a program object into *Pan I* by visiting a file whose text is the text of the program; to create a new program, she visits a non-existent file with an appropriate name. When the file is initially visited, the system creates a default view (the *primary view*) that shows the program text and on which analysis and structural navigation, as well as textual edits and navigation, can be performed. The user can choose between having the primary view formatted automatically according to some appearance specification and having

it retain the format in which it was entered. The command Reformat-Toggle[1] enables and disables primary view reformatting. The first time it is enabled, the user will be prompted for the name of a file containing a front end description specifying the desired appearance. When primary view reformatting is enabled, it is performed each time the program is analyzed.

The user can also add PPP views, called *alternate views*, using the *Pan I* command Add-Ppp-View. She will be prompted for the names of two files, one containing a front end description and one containing a back end description. After these are specified, a new view is created. Its initial contents are the formatted text corresponding to the state of the tree at the time it is added. Whenever the program is parsed, the view will be updated to reflect the structural changes. The view can be navigated textually, but cannot be edited or navigated structurally. The user can add as many views as she likes, using whatever front and back end descriptions she likes.

2.2 Elision

The user may designate certain program views as *elidable*. The commands Enable-Elision and Disable-Elision turn elision on and off for the current view. Elision is not permitted in the primary view, since the primary view is the source from which the program tree is created; elision in this view would destroy part of the program. To elide the subtree rooted at a node, she positions the *Pan I* tree cursor on that node in the primary view using the tree cursor motion commands (Cursor-To-Token, Tree-Up, etc.) and invokes the command Elide. This elides the node in all elidable views. The standard presentation of an elided subtree is a horizontal row of three equally-spaced dots; however, in the future, other back end descriptions may provide other kinds of elision, including elision which replaces code with system-generated summary information. To unelide a node, the user again positions the tree cursor on the node, and invokes Unelide. This unelides the node in each elidable alternate view. Unelision replaces the text that had been standing in the subtree's place with its current textual representation, accounting for any structural, semantic, or formatting changes since the node was elided, including elision and unelision of its descendants. Note that eliding or un-

[1]All PPP commands are currently bound to key sequences. Menu bindings could be added, either as the system default or in individual users' *Pan I* configuration files.

eliding a descendant of an elided node has no visible effect until all elided ancestors of the node have been unelided, that is, until the node's yield is visible.

3 Descriptions

There are three levels of description in PPP. The simplest, most concise, and least powerful is the status level. It is useful for fast generation of appearance specifications for new languages. The system translates it into the next level, the front end level, where it can either be used as-is or modified to suit the user's tastes. The front end level is a general language for describing formatting based on syntax. Finally, the back end level allows the user to provide portability and to make use of *Pan I*'s facilities for annotating programs and performing computations on them.

Status and front end descriptions are language-specific, and describe program appearance in terms of appearance primitives; back end descriptions are language-independent and describe how to implement appearance primitives. In other words, a front end or status description says what a program should look like; a back end description tells you how to get it to look that way. Both a back end and a front end (or status) specification are needed to fully specify a view of a program. (Standard back ends are provided for users who do not want to write their own.) Front and back ends can be mixed and matched freely; this provides conciseness, flexibility, and portability.

All the examples in this section are specifications for the appearance of the little language **asple**. The *Ladle* specification for **asple** is attached as Appendix D.

3.1 The Status Level

The status level is based on a simple idea. We assign one of four formatting statuses to each lexeme and abstract non-terminal in the language. (Grammar symbols and rules are known collectively as *operators.*) This status determines how it is presented on the screen. An operator's status may be one of punctuation, word, sentence, or paragraph. A punctuation operator is not given any special spacing; a word is separated from whatever surrounds it (except punctuation) by a single space on either side; a sentence is placed on

its own line, and nesting of one sentence within another is shown by deeper indentation of the nested sentence; a paragraph is separated from words or sentences by a single newline and from another paragraph by a double newline. Nesting of a paragraph within a sentence or paragraph is shown by indentation. Given these rules, a status for each operator in the language, and a textual representation for each operator in the language, we have a complete appearance specification for any program in the language.

A status specification consists of zero or more COMMON LISP lists. (A status specification for **asple** is found in Figure 5, and an **asple** program printed in accordance with this specification is found in Figure 6.) The first item of each list is one of `:punctuation`, `:word`, `:sentence`, and `:paragraph`. The remaining items of each list are the string names of operators in the language. The status of an operator is the status at the head of the last list it appears in. If the status is not specified, a lexeme has status `:word` and a non-terminal has status `:sentence`.

In the example, `loop` is not listed since we wish it to have the default status for a non-terminal, that is, `:sentence`. `","` is listed since we wish it to be treated as punctuation rather than as a word, the default for a terminal.

The status system is highly constrained: the set of statuses and the treatment of each status are fixed. This makes the system very rigid, but also makes it concise and easy to learn and use. A more extended version might be of interest, but it would probably also be more complicated to learn and use.

3.1.1 Creating and Processing Status Specifications

For help in creating a status description, a user can invoke the command Status-Helper from within *Pan I*. This command prompts the user for a file name and then creates a file with that name containing a list of all the operator names for the current language. The user can then edit this file to produce a status specification. The command Process-Status can be used to transform a status specification into a front end specification. When invoked, it prompts the user for two filenames. The first file is a status specification; the command creates a file with the second name containing a front end description corresponding to the given status specification. The user may then either process and use this description as described in Section 3.2 or edit it to modify particular cases she wishes to have handled differently.

```
(:punctuation "," ";")
(:paragraph "stmts" "decls" "program")
(:word "ANNOTATE" "ref" "plus" "times" "equals"
       "not_equals" "mode" "idlist""expression")
```

Figure 5: A Status Specification for Asple

3.2 Front End Specifications

A front end specification is a mapping from lexemes and abstract grammar rules to rules in the *Annotation Language* (*AL*). Therefore, to describe the front end descriptions, we must first describe the annotation language.

3.2.1 The Annotation Language

A rule in the annotation language is a list of annotation language items. (See examples in Figure 8.) An item is made up of an annotation type and a list of arguments.

The yield of a node of the abstract syntax tree is defined to be the concatenation of the yields of the items in the annotation list for that node. An item with the special annotation type `:child` is a placeholder for the corresponding child of the node being printed; its yield is the yield of the child. The yield of any other item is defined by the back end being used. `:child` is called a *system annotation type*, because its definition is fixed by the PPP kernel; all other types are called *user annotation types* because they are defined by the user in the back end. User annotation types might include a `:string` type with two arguments, a string and a font, and a `:space` type with an integer argument indicating the length of the piece of white space to be produced.

The AL serves as an intermediate representation for program appearance. Each AL item is a concise description of a piece of appearance that is in some sense primitive, for example, a single string, a group of spaces, or a marginal note. It is a low-level appearance specification which is independent of the eventual output representation, just as an intermediate representation in compilation is a low-level machine-independent specification for computations. A back end describes how to implement each appearance item in a particular representation. A combination of a front end description and

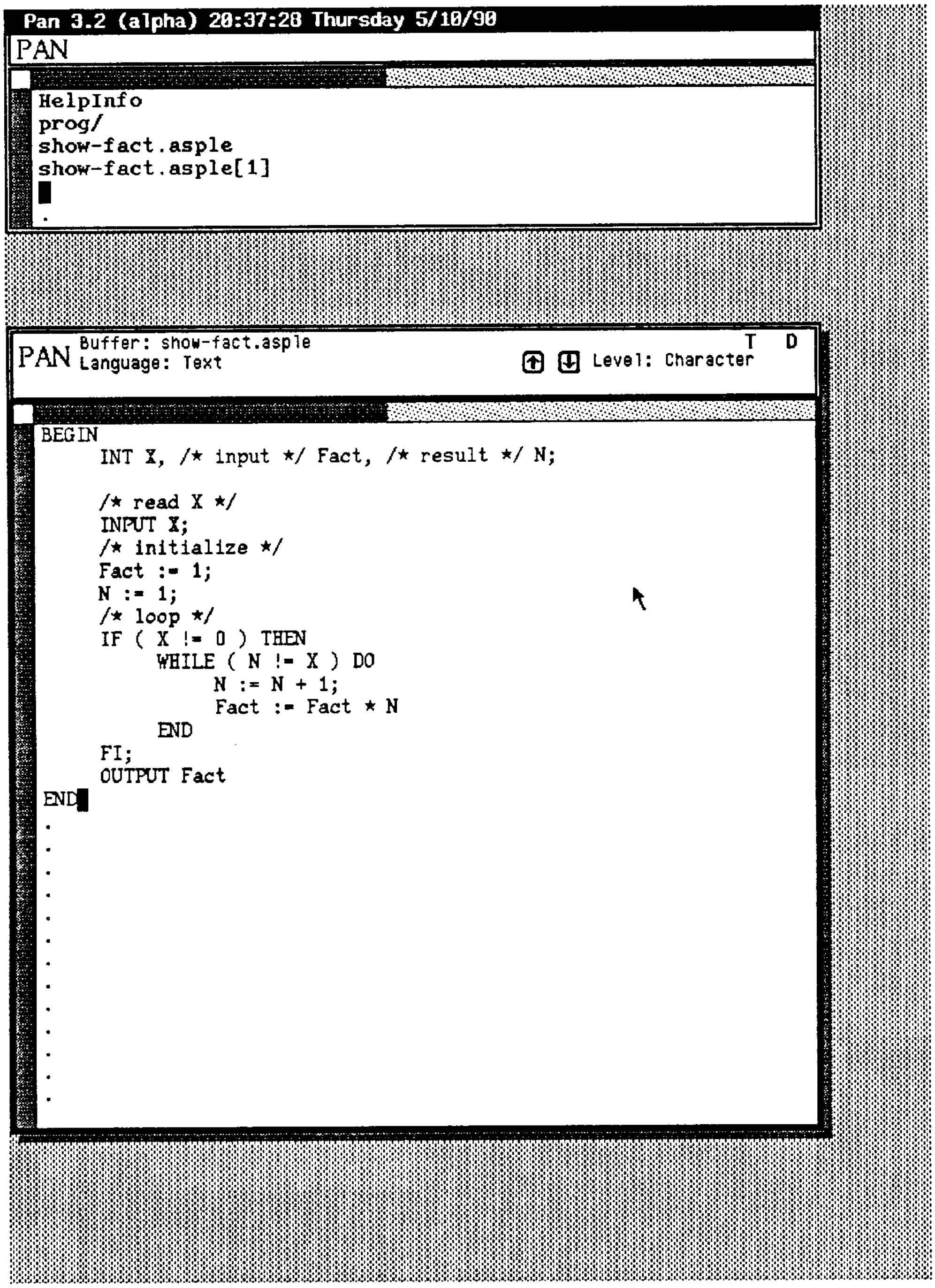

Figure 6: A Program Printed Using a Status Specification

$$
\begin{aligned}
\textit{annotate-list} &\rightarrow (\textit{annotate-item}^{*}) \\
\textit{annotate-item} &\rightarrow (\textit{keyword argument}^{*})
\end{aligned}
$$

keyword is any COMMON LISP keyword.
argument is any COMMON LISP value.

Figure 7: A Grammar for the Annotation Language

```
((:emstring "if"))

((:child) (:sp 1) (:child) (:sp 1) (:child) (:cr)
 (:in 1) (:child) (:cr)
 (:child))

((:colored-string :blue "var"))
```

Figure 8: Some Rules in the Annotation Language

a back end description describes how to present an appearance in a given representation. The ability to use different back ends with a single front end allows us to present the same appearance in a number of representations. It is this interchangeability of back ends that makes PPP portable.

Because user annotation types are defined by the back ends that support them, any keyword can be added to the set of annotation types. A user who wishes to add some new kind of appearance item need only assign a conventional meaning and argument signature to an appropriate keyword and provide a back end function to support it. By doing so, she has extended the AL.

The meaning of any user annotation type is determined by the back end in use. However, each annotation type has a fixed meaning which is adhered to in most back ends. Users who do not wish to use special back ends may use the system back ends and rely on the annotation types having the conventional meanings described here.

Annotate types other than `:child` supported by existing back ends are:

`:sp` Takes an integer argument n; prints n spaces.

`:cr` No arguments; prints a carriage return and indentation white space according to indent level.

`:self` No arguments; prints the string name of the node.

`:string` Has a string as an argument; prints the string.

`:in` Has an integer argument *depth*; tabs in by *depth* tabwidths.

`:comment` No arguments; prints the text representation of the comment attached to the current node.

`:emstring` Has a string as an argument; prints the string in an emphasized font.

`:startprog` No arguments; lets the back end know where the beginning of the program is. In many back ends this is a no-op; in a TEX back end it prints TEX prologue instructions.

`:endprog` No arguments; analogous to `:startprog`.

`:elide` No arguments; prints three dots.

3.2.2 Front End Descriptions Defined

A front end description maps node types to annotation language rules. A given node will always be printed with the rule associated with its node type. This restriction to one presentation per node type is common in the literature [Hor81] [Mat83] [Opp80] [Rub83], and was chosen to simplify the specification language. However, it would not be difficult to extend the system. Rules could be be chosen conditionally on the basis of the value of an arbitrary expression over syntactic and computed properties of the node. For example, the rule chosen to present a node representing a reference to a variable could depend on the type of the variable. Also, the system could provide commands to allow the user to interactively choose the rules to be used to present a node.

A front end description is a series of front end items, each of which is a COMMON LISP list. (Sample rules from a front end description for **asple** can be found in Figure 9.) A front end item starts with a keyword that tells what the rest of its parts must be.

- `:lexeme` must have an operator name and an AL rule
- `:commented-lexeme` must have a list of the operator name and the literal `"comment"`, and then an AL rule
- `:rule` must have an operator name, a right-hand side, and an AL rule
- `:commented-rule` same, but the right-hand side will include `"comment"` and the AL rule will be used to format the rule when it has an associated comment

The right-hand side is the list of operator names of the right-hand side of the grammar rule corresponding to the operator. In effect, it tells what the node's children are. The code that processes the descriptions does not read the right-hand sides; it reads only the keywords, the operator names, and the AL rules. The right-hand sides are provided for the user, to make the description more readable.

Note that in primary buffer reformatting, the front end specification is used only to describe whitespace format; any other directives will be ignored. This is because primary buffer reformatting is whitespace-only, to preserve the property that the program tree can be reproduced from the program text.

```
(:lexeme "IF" ((:emstring "if")))

(:lexeme "FI" ((:emstring "fi")))

(:lexeme "id" ((:self)))


(:rule "if_then" ("IF" "expression" "THEN" "stmts" "FI")
((:child) (:sp 1) (:child) (:sp 1) (:child) (:cr)
 (:in 1) (:child) (:cr)
 (:child)))

(:commented-rule "if_then"
("IF" "expression" "THEN" "stmts" "FI" "comment")
((:comment) (:cr) (:child) (:sp 1) (:child)(:sp 1) (:child) (:cr)
 (:in 1) (:child) (:cr)
 (:child)))
```

Figure 9: Excerpt from a Front End Description

3.2.3 User Commands Dealing With Front Ends

The command Create-Description prompts the user for a file name and creates a file with that name containing a front end description for the current language with all the AL rules empty. This provides the user with a framework that provides all the keywords, operator names, and right-hand sides; she then has merely to add the AL rules.

The system does not actually use rules in the form they are viewed and edited by the user, but rather uses an internal form. A user can create an internal-form file from a front end description file with the command Process-Description. By convention, a front end file is identified with the suffix ".desc" an internal form file is identified with the suffix ".int".

3.3 Back Ends

The program presenter itself defines the meaning of the annotation type `:child`. Other annotation types are defined by *back ends*. A back end is a mapping from annotation types to implementation functions. Back end specifications provide the user with three kinds of power. First, she can implement particular annotation types differently, as in the earlier example which used TeX commands to provide the emphasis for `:emstring`. Second, she can access *Colander* values and either display them or use them to influence printing. Third, she can include arbitrary COMMON LISP code. Used in conjunction with special front end and *Colander* specifications, she can provide interesting functionality. The TeX presentations and the elision mechanism are implemented in whole or part using back ends, and represent only two examples of the wide range of possibilities. This power comes at a cost, of course; users not familiar with COMMON LISP and the *Pan I* extension language will not be able to specify back ends, and users not familiar with *Colander* will not be able to use their full power. Fortunately, status descriptions and front end descriptions provide sufficient power for the casual user.

A back end is a mapping from annotation types (COMMON LISP keywords) to COMMON LISP functions. (Excerpts from a back end description can be found in Figure 10.) A back end is created using the macro define-back-end; it takes one argument, a COMMON LISP symbol that is to be the name of the back end. New functions are added to the back end using the

```lisp
;;;; This back end implements the functions
;;;; self
;;;; string
;;;; sp

(define-back-end basic)

(defun self-handler (tnode)
  (let ((region
         (Insert-Region (Create-Region-From-String
                          (string-for-tnode tnode))
                        :no-copy)))
    (cons (region-first-item region) (region-last-item region))))

(defun string-handler (tnode string)
  (declare (ignore tnode))
  (let ((region
         (Insert-Region (Create-Region-From-String
                          string)
                        :no-copy)))
    (cons (region-first-item region) (region-last-item region))))

(defun sp-handler (tnode n)
  (declare (ignore tnode))
   (let ((region (Insert-Region
                   (Create-Region-From-String
                      (make-string n :initial-element #\space))
                   :no-copy)))
    (cons (region-first-item region) (region-last-item region))))

(add-back-end-function basic #'self-handler :self)
(add-back-end-function basic #'string-handler :string)
(add-back-end-function basic #'sp-handler :sp)
```

Figure 10: Excerpts From a Back End Definition

macro add-back-end-function; its arguments are the back end being added to, the function being added, and the annotation type with which it is associated.

A back end function will be called with, as arguments, the tree node it is printing a piece of yield for and the arguments to the annotation item. Thus, it should take one more argument than the number associated with the annotation type being implemented. It should perform Pan text-editing operations to produce the yield of the type applied to the given arguments at the current cursor location, and it should return a pair of Pan text pointers delimiting the yield of the item. If the item has no yield, it should return the pair `(nil . nil)`.

In addition to using COMMON LISP code and the Pan extension language (a set of COMMON LISP functions that provide operations on Pan objects such as text and trees), back end functions can use the function getvar. Its arguments are a variable name and a tree node. It returns the value of that *Colander* variable at the given tree node.

Variables respected by existing back ends are:

`indent-incr` When its value is i, causes each child to have an indentation i character positions greater than its parent.

3.3.1 Uses for Back End Specifications

The back end mechanism was designed initially to provide portability between different output representations. The TEX versus Pan text example given in Section 1.6 is an example of this use. However, the back end mechanism turns out to provide a wider range of flexibility.

One simple use is to change the meaning of a single annotation type. A back end producing double-spaced views could be written easily; it would be the same as a standard back end, except that it would produce two carriage return characters for each `:cr` item, not just one. This is not really a new output representation, but still stays well within the original intention that back ends would serve to provide straightforward implementations of appearance primitives.

More complicated applications of back ends make use of their procedural nature, their access to the full power of COMMON LISP and the *Pan I* extension language, and their access to *Colander* values. The simplest instance of back end access to *Colander* values is for indentation. To have indentation reflect nesting depth, the user writes *Colander* rules which compute

the desired indentation for each node. The back end function for the `:cr` annotation type obtains this value from *Colander* using `getvar` and prints a carriage return and an appropriate indentation for the new line.

As another example, the implementation of elision makes use of the back end. The system code that elides a node simply replaces its AL rule with an `:elide` item whose argument is the former rule, sets its *Colander* extrinsic property `elide-node` to `t`, deletes its old yield, and calls the program presenter to print its new yield. Note that it needs to do work only at the root of the elided subtree. To unelide a node, the code simply restores the previous annotation rule, deletes the elided form of the yield, removes the *Colander* property, and calls the program presenter to print the new yield of each node in the subtree. There is a back end function supporting the `:elide` annotation type; it presents the node as a row of dots. Every back end function uses `getvar` to see whether the node it is called at is in an elided subtree before printing anything. If it is, the back end function prints nothing; otherwise, it prints the node as usual.

Being able to implement elision in the back end had two valuable results. First, it made it possible to add significant new functionality without changes to the program presenter's kernel. Second, it showed that clients can use the program presenter to provide special kinds of appearance not supported by the front end and to change program appearance between parses on the basis of arbitrary computations.

It will also be possible to use the back end to display semantic annotations. Suppose we wish to have code that annotates certain tree nodes with profile data. We can then have a front end description that adds a `:profile` item to the rules for certain nodes. The back end implementation of `:profile` can check for the existence of profile data for the current node, printing the data in an appropriate form and position if it exists, and printing nothing if it does not. Another possibility would be to have an annotation type `:var-name`. A standard back end could simply produce the string name of the variable, but a special back end for use on color machines could be used to examine the node's *Colander* properties, and print it in a special color if the variable is undeclared.

These examples hardly exhaust the power of the back end mechanism, but they begin to demonstrate its potential.

4 The Algorithm

Within a view, each node has a *direct yield*, the white space and text that arise directly from that node and not from its children. Suppose an if-then node represents the grammar rule *if-then* → **if** *expression* **then** *statement* . It might be printed using the following rule:

```
((:child) (:sp 1) (:child) (:sp 1) (:child) (:cr) (:in)
(:child))
```

The space, indent, and carriage return are part of the direct yield; the yields of the children are part of the *yield* of the node, but not part of its direct yield. (Note that in the current implementation of *Pan I*, each terminal has a node in the program tree; thus, the keywords are children. In a more abstract tree, there would be no nodes for the keywords, and they would be part of the direct yield.) A maximal contiguous piece of direct yield is called an *item* of direct yield; thus, all of the direct yield between the yields of two children, or before the first or after the last child, is one item. The `(:cr)` and `(:in)` annotate items make up a single item of direct yield. Note that no two items, for a single node or two different nodes, can possibly overlap.

After a parse, the parser delivers lists of nodes that have been inserted or deleted. First, the kernel processes deletions. To delete a node, we delete each item of its direct yield. This is easy, because for each node, for each view, the kernel maintains a list of pairs of pointers to the beginning and end of each item of the direct yield.

Insertions are processed in two phases. First, the kernel marks each inserted node with the AL rule to be used in its printing. Second, it inserts the text for the node's direct yield between the yields of its children. For each non-`:child` item in the AL rule, it *dispatches* the tree node and the item's arguments to the back end function associated with the item's annotation type. It stores the pointer pairs (returned by the back end) that record the locations of the direct yield items. Keeping pointers to items works well because it is easy to locate where to print an inserted node (before the next item of its parent's direct yield) and it is easy to delete a node (run down the list of items, deleting each one).

In general, nothing need be done with a node that is neither inserted nor deleted, even if nodes near it change. It is still a node for the same rule, and it is this rule that determines its direct yield. Text around its direct yield or between items of its direct yield may need to change, but these changes will

be performed when the nodes whose direct yields they are in are processed. This may not be the case with a sequence node, though. It may have children added or deleted, in which case it needs to have either more or fewer items of direct yield. This is handled by acting as if every sequence node with inserted or deleted children was both inserted and deleted; thus, we delete and recompute its entire direct yield. This is not as efficient as we would like, but gives a clean implementation. (The problem is that a sequence node whose length changes is treated as unchanged by the parser; PPP treats it as changed, but is unable to distinguish it from a node with children inserted or deleted whose length does not change, and thus performs redisplay more often than necessary.)

The incrementality model above is also a problem when errors are introduced into a program. Suppose the user edits a well-formed program, transforming it into an ill-formed program. What should the direct yield of an error node be? It must have some kind of direct yield; otherwise, its children will collide with each other without any space between them. The solution currently implemented simply inserts a carriage return between each pair of children.

A better solution would be to leave whatever space was between them before. This could be done as follows: When it is time to delete an item of direct yield, we first check for the case where a child that is next to the yield is not an inserted node and has a parent that is an inserted error node. In this case, the item is not deleted. This heuristic works reasonably well; it would work better with better node re-use.[2] An attempt was made to implement this solution, but, unfortunately, the parser discards some necessary information about deleted nodes.

There are other possible approaches to error node presentation; they are described in Section 5.

Reformatting of the primary view is done differently. The goal of reformatting is to make the text reflect a set of tree changes computed by the parser from a set of textual changes to the primary view. Thus, most of the textual changes corresponding to the structural changes have already

[2]Primary-buffer reformatting causes some nodes not to be re-used that otherwise might be. When the primary buffer is reformatted, the text of the program in the primary buffer is edited. These edits affect only white space, and should not change the tree. However, some of the tree's pointers to the program text point to white space, and invalidation of these pointers can prevent re-use of the nodes they appear at.

been made in the primary view. Only white space changes still need to be performed here. (Throughout this discussion, for "white space" read "text not significant to the parser".) So, we mark any *white space items* (maximal sequences of white space characters) that may have been corrupted as dirty. We take every dirty white space item and compute what should come between the two subtrees on either side. If it is different from the item, the item is deleted and the new text is inserted. Currently, we use a heuristic that says every white space item is dirty. Work has been done on another heuristic, but it is incomplete.

5 Further Work

There are two categories of further work: enhancements to the system which would improve its functionality but would not constitute new research, and new research areas.

5.1 Additional Research

There is interesting work to be done in the part of the formatter that assigns printing rules to nodes. For example, it would be easy to take some of the tree pattern-matching code developed at Berkeley [Far88] and use it for this purpose. Because rule assignment is isolated from printing, it would be easy to experiment in this area. We could also add declarative access to and dependence on semantics; these are already procedurally available.

A lot of interesting work remains to be done on ways of presenting syntactically incorrect code. For example, the status system provides obvious heuristics for what to produce as the direct yield of an error node — line breaks between sentences, spaces between words, et cetera. An interesting question is how the results of a policy which computed error node yields based on a heuristic would compare to a policy that let the user's text remain in place when it was recoverable.

It would be useful to be able to incrementally reformat views based on changes to the front end specification. For example, when the user changed a particular rule, only nodes whose appearances were affected by the change would be reformatted. This would be a useful environment for developing and debugging front end specifications.

Finally, now that this testbed has been developed, there is room for research on what program presentation styles are most useful to programmers.

5.2 System Enhancements

Current presentations are constrained not to re-order a node's children. However, a user might want this functionality. For example, expressions could be printed in prefix, infix, or postfix notation. Adding this would require a slight increase in the complexity of both the descriptions and the printing algorithm, but it would increase the system's usefulness as a tool for experimenting with program appearance, and is probably worth the cost.

Sequence nodes should be handled more incrementally; this may require major changes to the parser, the presenter, or both. It would also be good to come up with better heuristics for what whitespace in the main buffer needs to be recomputed.

Structural navigation of alternate views should be added. This would require creating and maintaining pointers from the text of alternate views to the tree. It may not be possible to simply re-use the existing (primary view) structural navigation code for alternate buffers, as pointers from tree to view are different in the two cases.

It would be convenient for users to be able to combine PPP specifications and PPP-relevant *Colander* specifications. The *Colander* portion could be in either *Colander* syntax or some PPP-specific syntax. The PPP description processors would extract the *Colander* specification and pass it on to *Colander* for preprocessing.

6 Known Bugs and Problems

PPP's implementation is fairly sound, except for its handling of comments.

Before the PPP project was initiated, *Pan I* ignored comments beyond the lexical level. That is, a comment would be part of the lexical stream maintained for a program, but it would not be part of the program tree. This is sufficient for traditional program analysis, which ignores comments. It is not sufficient for program presentation, since we need to know what a comment applies to before we know where to display it. Accordingly, PPP has an initial phase which uses a heuristic to determine what program node

each comment applies to, and attaches each comment to the relevant tree node as an annotation. One problem is that the heuristic is too simple.

There is also an awkwardness in the way comments are handled in the front end specification language. Commented nodes are treated as a special case in the front end descriptions; each operator is given two AL rules, one to be used in the presence of comments and one to be used in their absence. Since this decision was made, it has become clear that the same effect could be achieved by giving each operator one rule, some of whose items would have back end implementations that produce output if and only if there is a comment on the node. This would make descriptions more concise and simplify the code that processes descriptions and attaches rules to nodes.

If a semantic value that affects the presentation changes, but the node it is at does not change syntactically, the presenter will not notice. It is possible to add *Colander* triggers that will notify the PPP kernel of these changes. Nodes where such changes occur can be reformatted by treating the changes as a deletion of the node followed by an insertion of the node.

Reformatting the main buffer marks the tree as dirty and causes a lot of re-computation. Since these edits are known not to change the tree, they should not invoke a reparse. Fixing this would involve having the program presenter maintain correspondences between text and tree initially set up by the parser. Another possibility is to create a lexer call which will restore the correspondences, working on the assumption that only whitespace changes have been made.

7 Related Research

This section puts PPP in perspective in the context of research on program presentation. There are six major areas or problems of interest in the field:

1. specification languages for describing program appearance
2. line breaking algorithms
3. incremental algorithms
4. how program presenting can work in an interactive editor environment
5. what kind of appearances are useful to programmers

6. elision

PPP embodies interesting new work in the area of specification languages. It has entirely disregarded the question of line breaking, the primary focus of many papers published under the rubric of "pretty printing." It presents one solution to the problems of providing program presentation incrementally in an editor environment. It provides a useful test bed for investigating kinds of program appearance, since it is general and flexible. Many papers in the literature describe particular elision policies and algorithms for implementing them. PPP, rather than implementing a particular policy, supports elision of subtrees and allows client code to choose the subtrees to be elided. Thus, as with other appearance issues, it will serve as a useful test bed for policy development.

7.1 XP

Some of the earliest work in program formatting was done for LISP systems. For example, Joel Moses's 1967 PhD thesis [Mos67] refers to a LISP pretty printer developed by Whitfield Diffie. LISP pretty printers have developed over the years; a current system of particular interest is XP, a pretty printer for COMMON LISP[Wat89]. It provides functions that replace the standard COMMON LISP printing functions. It prints COMMON LISP functions and data structures in a way that reveals their structure and can do sensible line breaking if the material to be printed exceeds the line width available. It is configurable at several levels. There are a number of parameters affecting printing that can be set by the user, for example, the maximum number of rows and columns to be used in printing an object. There are also new `format` directives that allow the user to specify the preferred line breaking and indentation of the material being printed. For example, it is possible to group parts of a format string together, specifying that breaks outside the group should be preferred to breaks within the group, and to indicate good potential line breaks. Finally, it is possible to define to the system what printing function should be used to print objects of a particular type (these functions can of course be defined with the special `format` directives provided by the system). Thus, it is possible to use the default printings for most objects, but override them for certain objects and add new printings for new object types defined by the user. The same formalism could clearly

be applied to languages other than LISP, since anything that handles nested LISP lists translates naturally to trees.

This formalism is extremely powerful, since printing descriptions can include arbitrary LISP code. PPP can also do this. In each case, the escape to LISP is handicapped in a different way. In XP, because there are no back pointers from children to parents, making printings depend on inherited context will be tricky. In PPP, because LISP code is called in individual AL items, the LISP code for presenting a single node will be scattered among the functions for the different items. If the escapes to LISP are not used, and we use straightforward format statements in the XP case and front end descriptions with a standard back end in the PPP case, the two formalisms are equivalent in power, except that PPP provides the more concise status formalism and XP provides line breaking directives. Recall, however, that it would not be difficult to use PPP to produce input to a text-processing system that could do line breaking.

My primary reservation with the XP model is that, while `format` as extended by Waters is a powerful descriptive method, it is not friendly for novices. This is not a problem in the XP context, where printing functions are already provided for anything a novice might want to do, since the language to be presented is already defined; however, in the PPP context, where users may be defining new languages, there needs to be some easy way like the status system to get descriptions for new languages.

In many ways, XP and PPP address different problems. XP addresses line breaking in a batch-printing ascii environment; PPP addresses presentation without line breaking in an incremental, interactive environment with extended ascii — for example, it can be used to support different fonts. Both are powerful; each is tailored to its particular setting.

7.2 tgrind and vgrind

The UNIX tools tgrind and vgrind use a model of program presentation different from that used in PPP and other work in the area [Jac87] [PJ89]. The program to be presented is purely a text stream without more than lexical structure and the presenter acts as a filter, adding some typesetting information and annotations but not altering the whitespace layout.

The presenter does not parse the program, and the description does not include a complete grammar of the language. Rather, the presenter acknowl-

edges a hardwired set of programming language constructs, such as keywords, comments, string literals, and function names. It is not possible to describe or specify treatments for constructs outside this set. The user specifies, on a per language basis, ways of recognizing these constructs, usually by beginning and end markers and keyword lists. The presenter treats each construct in a fixed way — e.g., puts keywords in boldface — and adds certain annotations, such as page headers, marginal function names, and line numbers. White space and indentation are typically not affected, except perhaps for some additional space around function headers and the like; thus, program structure is not reflected by the display.

The grind approach has two kinds of limitations, only one of which can be addressed without a complete paradigm shift. The first is its hardwired set of programming language constructs and hardwired treatments of them. These could be replaced with the ability to specify syntactic classes and associated treatments.

The other limitation is more serious. The text-filter model does not allow for reflecting general syntactic structure, let alone context of any sort. Also, it makes generation of formatted text from a tree, increasingly important in programming environments, impossible.

All that said, I do not want to neglect the strengths of the grind model. It is simple to write the presenter and the language descriptions, and the system addresses fonts, headers, and other advanced display features that are very useful in making code readable and that are ignored in many supposedly more advanced systems. These grind-type program presenters are especially effective when combined with either "hand-made" whitespace (which programmers have traditionally provided on a routine basis) or other whitespace facilities, for example in an Emacs language mode. This model is an excellent example of a good "eighty percent solution," the twenty-percent effort that provides eighty percent of the needed result. However, it is not of any use in an editor context, where we are working from a program tree and our primary interests are syntactic and semantic, not lexical.

All the other solutions I have examined do include consideration of program syntax, and are thus more closely related to the problems addressed by PPP.

```
statement =  'IF' expression 'THEN'
                   stmts
             'ELSE'
                   stmts
             'FI'
=> if_else
```

Figure 11: Specification by Grammar Layout

7.3 Specification Languages

Many papers in the literature are addressed primarily to questions of line breaking [PS87] [Opp80]. These papers also generally include a specification scheme consisting of parse rules decorated with directives. These schemes differ primarily in terms of what directives are available, which depends on the model of line breaking. Babel [Hor81] also uses a decorated grammar scheme. It provides a small, fixed set of directives: carriage return, increase indentation, and decrease indentation. It does not support line breaking. PPP's front end descriptions are equivalent to a decorated grammar scheme, except that they are unique in the extensibility of the set of directives available.

An interesting variation on the decorated grammar scheme is to present the grammar rules as one would wish the code produced by them to be presented. For example, we could specify an appearance for the **asple** if-then-else statement by printing its *Ladle* grammar rule as in Figure 11. This is the approach taken in the Ada language specification [Uni80]. It has the advantage of being pleasing to the eye and easy to understand. However, it is somewhat limited. It does not easily express line breaking directives, and there is no easy way for a reader to see whether a given piece of white space should be represented by a tab or a series of spaces.

Rubin [Rub83] also includes a specification scheme involving a decorated grammar, and adds a form of abbreviation: if some grammar symbol is always followed by the same pretty printing directive, the association between symbol and directive can be specified once and will be obeyed throughout. For example, suppose we were to implement this scheme in the PPP front end. If, in a particular front end description, every occurrence of the grammar

terminal "integer" was followed by `(:sp 1)`, we could specify this once, and omit the `(:sp 1)` in each rule that referred to an integer. This seems of limited utility, however; each grammar rule still has a presentation rule, but not all of its eventual appearance is clear from the rule alone. This form of description seems likely to be confusing. Of course, carried to its logical conclusion, where each grammar symbol is given a single treatment to be used in all contexts, we arrive at something like the status system.

The Cornell Program Synthesizer Generator [RT85] also uses a description scheme involving a decorated abstract grammar. This scheme is specifically designed for use in specifications to an editor, and includes information especially for browsing and editing issues. In particular, it is possible to specify whether a particular tree component is editable and whether it is structurally selectable. It is also possible to specify alternate presentations for a given production; the user is then able to toggle between the different options. Among other things, this can be used to provide user-controlled elision. This scheme does not provide for the use of alternate output representations, access to semantics, or any form of abbreviation, all of which are provided within PPP. This extended form of elision, with multiple alternate presentations, could be supported in a way similar to that used to support the more limited form of elision provided in PPP.

The Mentor programming environment [MCC86] provides a powerful specification language, allowing reference to annotations and conditionality on tree structure. Rather than being associated with grammar rules, node presentations are associated with tree patterns; a node is printed using the first rule whose pattern matches at the node. Rules contain standard pretty printing directives, and may also contain conditional statements which dictate a particular presentation based on the tree's structure. This conditionality within the rules does not extend the power of the language beyond that provided by its pattern-matching capabilities, but it does make it possible to write more concise descriptions. PPP supports reference to annotations — see, for example, its handling of comments. Because of its modular structure, conditionality on tree structure could be introduced; see the discussion of tree pattern matching in Section 5.

Mateti [Mat83] comes up with a method of formally specifying the task of a program-formatting algorithm and rigorously proves the correctness of one algorithm.

7.4 Efficiency Concerns

Caplinger and Hood [CH86] present an algorithm addressing interactive systems' need for efficiency. Rather than being truly incremental, in the sense of creating a presentation that it then changes in response to program changes, their algorithm provides *partial presentation* — it will provide as much program text as is needed, possibly with elision, around a given point in the program. It is language-specific and embodies a particular presentation scheme and elision scheme rather than serving as a test bed. The particular appearance it provides for Fortran is derived in much the way that status-based schemes are. Caplinger and Hood include a survey stating that most structured program editors do not use incremental program presenters.

Cameron's abstract pretty printer [Cam88] uses an idea similar to the PPP concept of back ends. It is a program presenter that requires a fixed set of functions to access its output device. By supplying these functions, the user can support printing on any of a number of output devices, and can also support other tasks, such as mapping between tree and text. This model is more limited than PPP's because the front end description and the set of back end functions are both fixed, whereas in PPP, the front end is description-driven and the set of back end functions is expandable.

7.5 User Issues

Baecker and Marcus [BM86] provide valuable insights on how programs should be presented. With adequate back end support, the PPP framework could be used to support these presentations and for investigation of program presentation in general. While the front end descriptions are not powerful enough to express all possible appearances — lacking, for example, the ability to describe several alternate presentations for a single node type — the implementation of elision illustrates that it is possible to use PPP to display nodes based on AL rules assigned to nodes by other tools. The primary limitation of our model is that tree nodes are constrained to be presented in the order of their appearance in the tree. The other problem is that a given back end function is called on only a small piece of the appearance being generated; it is thus difficult for the back end to perform computations, such as page layout, line breaking, and page breaking, which operate over larger units of the appearance. Note that page layout computations, such as

positioning marginal notes, are necessary for the presentations proposed by Baecker and Marcus. The natural approach to providing these presentations using PPP would be to have the back end emit a layout specification which could be processed by an appropriate mechanism, as in the TEX example given in this paper.

Mikelsons [Mik81] presents a scheme for choosing code to be elided based on a notion of the user's current task. Since PPP allows client code to mark nodes for elision, it is capable of serving as a test bed for such schemes.

7.6 Viz and UAL

Garlan's Viz and UAL [Gar85] provide incremental display based on a powerful specification paradigm. A user can have multiple simultaneous views of a program. Within a view, the display of any particular tree node can be conditional on various values, including the amount of space available. Descriptions are parameterizable by general style parameters. The model of incremental display is much like PPP's. PPP has simpler computations to do because of its somewhat more restricted scope. It also has a different specification model, allowing more procedural specification. It will take some time with a user community to determine the wisdom of these particular design choices.

7.7 Summary

In summary, the solution generally posed in the literature to the problem of how to describe program appearance is to decorate grammar rules with printing directives, or to associate sequences of strings and directives with grammar rules. The latter solution, chosen by PPP, is somewhat more general, as it supports alternate concrete syntaxes, including different keywords. Some systems, such as the Cornell Program Synthesizer Generator, allow the user some control over which presentation is chosen for a given instance of a rule. Others provide declarative means for specifying that a presentation should display semantic information, or that the presentation chosen for a node should depend on such information. There may be room for more work in concise specifications, such as the status system, for use in the rapid prototyping of new languages, and on the best formalisms for expressing dependence on semantics.

In terms of algorithms, few systems are incremental, though there has been some work on partial presentations. There has also been some work done on retargetability. There has been a great deal of work done on line breaking algorithms.

Some work has been done on preferred appearances for reflecting syntax [BM86], but little has been done with semantically-powerful viewing.

PPP is unique in providing a combination of incrementality and retargetability and in providing an extensible set of directives. It is equaled only by XP in its provision of a range of levels for specifying appearance, and it works in a very different setting. Because of its flexibility, and its position as part of a sophisticated language-oriented editor, it will provide a valuable test bed for experimenting with program appearance.

8 Achievements

PPP provides *Pan I*'s first view of programs that is not simply the text as entered by the user. It provides multiple views of a single program tree, and is language-independent, description-driven, and incremental.

One of the project's major achievements is the development of the status system, a succinct and powerful formalism for describing program appearance.

The other major achievement is the three-part architecture and annotation language. This is an important innovation for two reasons. First, the annotation language serves as an intermediate representation for program appearances, so that a single appearance description can be used to produce several views. Second, it serves as an escape to procedural specification and has been used to support elision.

PPP also supports display of erroneous program text, a problem that, so far as I know, no other system addresses. I know of no systems that support syntactically incorrect programs in any interesting way.

The main design error was to make commented nodes a special case. Overall, however, the project has produced a powerful, flexible test bed for experimenting with structural viewing, formatted text, and semantically-powerful views of programs.

References

[Bal89] R. A. Ballance. Syntactic and semantic checking in language-based editing systems. University of California, Berkeley, PhD Dissertation, CS Report UCB/CSD89/548, 1989.

[BGV90] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The Pan language-based editing system for integrated development environements. To appear in ACM SIGSOFT '90 Fourth Symposium on Software Development Environments, 1990.

[BM86] R. Baecker and A. Marcus. Design principles for the enhanced presentation of computer program source text. *CHI '86 Proceedings*, pages 51–58, April 1986.

[But89] J. Butcher. Ladle. University of California, Berkeley CS Report UCB/CSD89/519, 1989.

[BV87] R. A. Ballance and M. L. Van De Vanter. Pan I: An introduction for users. University of California, Berkeley CS Report UCB/CSD88/410, 1987.

[Cam88] R. D. Cameron. An abstract pretty printer. *IEEE Software*, pages 61–67, November 1988.

[CH86] M. Caplinger and R. Hood. An incremental unparser for structured editors. *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 65–74, 1986.

[Far88] C. Farnum. A tree transformation facility using typed rewrite systems. University of California, Berkeley CS Report UCB/CSD88/431, 1988.

[Gar85] D. Garlan. Flexible unparsing in a structure editing environment. CMU Technical Report CMU-CS-85-129, 1985.

[Hor81] M. R. Horton. Design of a multi-language editor with static error-detection capabilities. University of California, Berkeley Electronics Research Laboratory Memorandum UCB/ERL M81/53, 1981.

[Jac87] V. Jacobson. TGRIND(1). UNIX programmer's manual 4.3 BSD, 1987.

[Mat83] P. Mateti. A specification schema for indenting programs. *SP&E*, 13:163–179, 1983.

[MCC86] E. Morcos-Chounet and A. Conchon. PPML: A general formalism to specify prettyprinting. IFIP, 1986.

[Mik81] M. Mikelsons. Prettyprinting in an interactive programming environment. IBM Technical Reports RC 8756, 1981.

[Mos67] J. Moses. Symbolic integration. MIT PhD thesis, 1967.

[OC90] P. W. Oman and C. R. Cook. Typographic style is more than cosmetic. *CACM*, 33(5):506–520, May 1990.

[Opp80] D. C. Oppen. Prettyprinting. *TOPLAS*, 2(4):465–483, October 1980.

[PJ89] D. Presotto and W. Joy. VGRIND(1). UNIX programmer's manual 4.3 BSD, 1989.

[PS87] W. W. Pugh and S. J. Sinofsky. A new language-independent prettyprinting algorithm. Cornell University CS Technical Report TR 87-808, 1987.

[RT85] T. Reps and T. Teitelbaum. The synthesizer generator reference manual. Cornell University Computer Science Technical Report, 1985.

[Rub83] L. Rubin. Syntax-directed pretty-printing — a first step towards a syntax-directed editor. *IEEE Transactions on Software Engineering*, 9(2):119–127, March 1983.

[Uni80] United Stated Department of Defense. Reference manual for the Ada programming language: Proposed standard document. Superintendent of Documents, U. S. Government Printing Office, 1980.

[Wat89] Richard C. Waters. XP: A Common Lisp pretty printing system. MIT AI Memo No. 1102a, 1989.

A A Front End Description

```
;; Copyright 1987-1990 by the Regents of the University of California.
;; All rights reserved.
;;
;; PPP Description:
;;
;; A front end specification for asple.  It can be used with
;; either the standard or the TeX back end.

(:lexeme "ERROR" ((:self)))

(:commented-lexeme ("ERROR" "comment") ((:self) (:sp 1) (:comment)))

(:lexeme "!=" ((:string "!=")))

(:lexeme "=" ((:string "=")))

(:lexeme ")" ((:string ")")))

(:lexeme "(" ((:string "(")))

(:lexeme "*" ((:string "*")))

(:lexeme "+" ((:string "+")))

(:lexeme ":=" ((:string ":=")))

(:lexeme "," ((:string ",")))

(:lexeme ";" ((:string ";")))

(:commented-lexeme ("," "comment") ((:string ",") (:sp 1) (:comment)))

(:commented-lexeme (";" "comment") ((:string ";") (:sp 1) (:comment)))

(:lexeme "TRUE" ((:emstring "true")))
```

```
(:lexeme "THEN" ((:emstring "then")))

(:lexeme "WHILE" ((:emstring "while")))

(:lexeme "REF" ((:emstring "ref")))

(:lexeme "OUTPUT" ((:emstring "output")))

(:lexeme "INT" ((:emstring "int")))

(:lexeme "INPUT" ((:emstring "input")))

(:lexeme "IF" ((:emstring "if")))

(:lexeme "FI" ((:emstring "fi")))

(:lexeme "FALSE" ((:emstring "false")))

(:lexeme "END" ((:emstring "end")))

(:lexeme "ELSE" ((:emstring "else")))

(:lexeme "DO" ((:emstring "do")))

(:lexeme "BOOL" ((:emstring "bool")))

(:lexeme "BEGIN" ((:emstring "begin")))

(:lexeme "id" ((:self)))

(:commented-lexeme ("id" "comment") ((:self) (:sp 1) (:comment)))

(:lexeme "integer" ((:self)))

(:commented-lexeme ("integer" "comment") ((:self) (:sp 1) (:comment)))
```

```
(:rule "idlist"
 ("id" ",")
((:child) (:child) (:sp 1)))

(:rule "stmts"
 ("statement" ";")
((:child) (:child) (:cr)))

(:rule "decls"
 ("declaration" ";")
((:child) (:child) (:cr)))

(:rule "program"
 ("BEGIN" "decls" ";" "stmts" "END")
((:startprog) (:child) (:cr)
(:in 1) (:child) (:child) (:cr) (:cr)
(:in 1) (:child) (:cr)
(:child) (:endprog)))

(:commented-rule "program"
 ("BEGIN" "decls" ";" "stmts" "END" "comment")
((:startprog) (:comment) (:cr) (:cr)
(:child) (:cr)
(:in 1) (:child) (:child) (:cr) (:cr)
(:in 1) (:child) (:endprog) (:child)))

(:rule "ref" ("REF" "mode") ((:child) (:sp 1) (:child)))

(:commented-rule "ref"
 ("REF" "mode" "comment")
((:child) (:sp 1) (:child) (:sp 1) (:comment)))

(:rule "plus"
 ("expression" "+" "expression")
((:child) (:sp 1) (:child) (:sp 1) (:child)))

(:commented-rule "plus"
```

```
 ("expression" "+" "expression" "comment")
((:child) (:sp 1) (:child) (:sp 1) (:child) (:sp 1) (:comment)))

(:rule "times"
 ("expression" "*" "expression")
((:child) (:sp 1) (:child) (:sp 1) (:child)))

(:commented-rule "times"
 ("expression" "*" "expression" "comment")
((:child) (:sp 1) (:child) (:sp 1) (:child) (:sp 1) (:comment)))

(:rule "ANNOTATE"
 ("(" "expression" ")")
((:child) (:child) (:child)))

(:commented-rule "ANNOTATE"
 ("(" "expression" ")" "comment")
((:child) (:child) (:child) (:sp 1) (:comment)))

(:rule "equals"
 ("(" "expression" "=" "expression" ")")
((:child) (:child) (:sp 1) (:child) (:sp 1) (:child) (:child)))

(:commented-rule "equals"
 ("(" "expression" "=" "expression" ")" "comment")
((:child) (:child) (:sp 1) (:child) (:sp 1) (:child) (:child)
(:sp 1) (:comment)))

(:rule "not_equals"
 ("(" "expression" "!=" "expression" ")")
((:child) (:child) (:sp 1) (:child) (:sp 1) (:child) (:child)))

(:commented-rule "not_equals"
 ("(" "expression" "!=" "expression" ")" "comment")
((:child) (:child) (:sp 1) (:child) (:sp 1) (:child) (:child)
(:sp 1) (:comment)))
```

```
(:rule "assignment"
 ("id" ":=" "expression")
((:child) (:sp 1) (:child) (:sp 1) (:child)))

(:commented-rule "assignment"
 ("id" ":=" "expression" "comment")
((:comment) (:cr) (:child) (:sp 1) (:child) (:sp 1) (:child)))

(:rule "if_then"
 ("IF" "expression" "THEN" "stmts" "FI")
((:child) (:sp 1) (:child) (:sp 1) (:child) (:cr)
(:in 1) (:child) (:cr) (:child)))

(:commented-rule "if_then"
 ("IF" "expression" "THEN" "stmts" "FI" "comment")
((:comment) (:cr)
(:child)  (:sp 1) (:child) (:sp 1) (:child) (:cr)
(:in 1) (:child) (:cr) (:child)))

(:rule "if_else"
 ("IF" "expression" "THEN" "stmts" "ELSE" "stmts" "FI")
((:child)  (:sp 1) (:child) (:sp 1) (:child) (:cr)
(:in 1) (:child) (:cr)
(:child) (:cr)
(:in 1) (:child) (:child)))

(:commented-rule "if_else"
 ("IF" "expression" "THEN" "stmts" "ELSE" "stmts" "FI" "comment")
((:comment) (:cr) (:child) (:sp 1) (:child) (:sp 1) (:child) (:cr)
(:in 1) (:child) (:cr)
(:child) (:cr)
(:in 1) (:child) (:child)) )

(:rule "loop"
 ("WHILE" "expression" "DO" "stmts" "END")
((:child) (:sp 1) (:child) (:sp 1) (:child) (:cr)
(:in 1) (:child) (:cr)
```

```
(:child)))

(:commented-rule "loop"
 ("WHILE" "expression" "DO" "stmts" "END" "comment")
((:comment) (:cr) (:child) (:sp 1) (:child) (:sp 1) (:child)(:cr)
(:in 1) (:child) (:cr)
(:child)))

(:rule "input"
 ("INPUT" "id")
((:child) (:sp 1) (:child)))

(:commented-rule "input"
 ("INPUT" "id" "comment")
((:comment) (:cr)
(:child) (:sp 1) (:child)))

(:rule "output"
 ("OUTPUT" "expression")
((:child) (:sp 1) (:child)))

(:commented-rule "output"
 ("OUTPUT" "expression" "comment")
((:comment) (:cr)
(:child) (:sp 1) (:child)))

(:rule "declaration"
("mode" "idlist") ((:child) (:sp 1) (:child)))

(:commented-rule "declaration"
 ("mode" "idlist" "comment")
((:comment) (:cr)
(:child) (:sp 1) (:child)))
```

B A Standard Back End

```
;;; -*- Mode: common-lisp; Package: ppp-std-back-end; -*-
;;; Copyright 1987-1990 by the Regents of the University of California.
;;; All rights reserved.
;;;
;;; PIPER:
;;;
;;; a simple PPP back end.


(provide "ppp-std-back-end")
(in-package "ppp-std-back-end")
(export '(back))

(require "extensions")
(require "version")
(require "presentations")
(require "ppp-ext")

(shadowing-use-package "extensions")
(use-package "text")
(use-package "ppp")


(version:def-file-id "ppp-std-back-end.cl"
  "@(#)ppp-std-back-end.cl 26.1  7/23/90  15:05:09\
  Copyright 1987-1990 by the Regents of the University of California")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;; A back end.  It respects/uses the variables
;;;; compute-indent-level
;;;; elided
;;;; and implements the functions
;;;; elide
;;;; self
```

```
;;;; in
;;;; string
;;;; sp
;;;; cr
;;;; emstring
;;;; startprog
;;;; endprog

;;;; Remember that a back end function's first argument is a tnode,
;;;; that it is called with the cursor appropriately positioned, and
;;;; that it must return a pair of pointers.

(define-back-end back)

(defun self-handler (tnode)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let ((region
             (Insert-Region (Create-Region-From-String
                              (string-for-tnode tnode))
                            :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun string-handler (tnode string)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let ((region
             (Insert-Region (Create-Region-From-String
                              string)
                            :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun in-handler (tnode depth)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let ((region (Insert-Region
                      (Create-Region-From-String
```

```
                        (make-string
                          (* depth (variable Pan-Prettyprint-Indent-Width))
                          :initial-element #\space))
                    :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun sp-handler (tnode n)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let ((region (Insert-Region
                      (Create-Region-From-String
                        (make-string n :initial-element #\space))
                      :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun cr-handler (tnode)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let* ((indent (or (getvar 'compute-indent-level tnode) 0))
             (region (Insert-Region
                       (Create-Region-From-String
                         (string-concat
                           (string #\newline)
                           (make-string
                             (* indent
                                (variable Pan-Prettyprint-Indent-Width))
                           :initial-element #\space)))
                       :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun comment-handler (tnode)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let ((region
             (Insert-Region (Create-Region-From-String
                              (string-for-tnode
                                (ppp::pprint-info-comment
```

```
  (otree:tnode-pp-info tnode))))
                              :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun startprog-handler (tnode)
  (declare (ignore tnode))
  (cons nil nil))

(defun endprog-handler (tnode)
  (declare (ignore tnode))
  (cons nil nil))

(defun emstring-handler (tnode string)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
    (let ((region
           (Insert-Region (Create-Region-From-String string)
                          :no-copy)))
      (region-set-font region 4)
      (cons (region-first-item region) (region-last-item region)))))

(defun elide-handler (tnode oldrule)
  (declare (ignore oldrule))
  (if (and (not (otree:tnode-is-abstract-root? tnode))
           (tnode-is-elided? (otree:tnode-parent tnode)))
      (cons nil nil)
      (let ((region (Insert-Region
                      (Create-Region-From-String ". . .")
                      :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun tnode-is-elided? (tnode)
  (cond
    ((otree:tnode-is-abstract-root? tnode)
     (unless (otree:tnode-is-error-NT? tnode)
       (getvar 'elide-node tnode)))
    (t (or (unless (otree:tnode-is-error-NT? tnode)
```

```
            (getvar 'elide-node tnode))
          (tnode-is-elided? (otree:tnode-parent tnode))))))

(add-back-end-function back #'self-handler :self)
(add-back-end-function back #'in-handler :in)
(add-back-end-function back #'sp-handler :sp)
(add-back-end-function back #'cr-handler :cr)
(add-back-end-function back #'comment-handler :comment)
(add-back-end-function back #'elide-handler :elide)
(add-back-end-function back #'string-handler :string)
(add-back-end-function back #'emstring-handler :emstring)
(add-back-end-function back #'startprog-handler :startprog)
(add-back-end-function back #'endprog-handler :endprog)
```

C A TeX Back End

```
;;; -*- Mode: common-lisp; Package: ppp-tex-back-end; -*-
;;; Copyright 1987-1990 by the Regents of the University of California.
;;; All rights reserved.
;;;
;;; PIPER:
;;;
;;; a PPP back end producing TeX source


(provide "ppp-tex-back-end")
(in-package "ppp-tex-back-end")
(export '(tex))

(require "extensions")
(require "version")
(require "presentations")
(require "ppp-ext")

(shadowing-use-package "extensions")
```

```
(use-package "text")
(use-package "ppp")

(version:def-file-id "ppp-tex-back-end.cl"
  "@(#)ppp-tex-back-end.cl 26.1  7/23/90  15:05:12\
  Copyright 1987-1990 by the Regents of the University of California")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;; A tex back end.  It respects/uses the variable
;;;; compute-indent-level
;;;; elided
;;;; and implements the functions
;;;; elide
;;;; self
;;;; in
;;;; string
;;;; sp
;;;; cr
;;;; emstring
;;;; startprog
;;;; endprog

;;;; Remember that a back end function's first argument is a tnode,
;;;; that it is called with the cursor appropriately positioned, and
;;;; that it must return a pair of pointers.

(define-back-end tex)

(defun elide-handler (tnode oldrule)
  (declare (ignore oldrule))
  (if (and (not (otree:tnode-is-abstract-root? tnode))
           (tnode-is-elided? (otree:tnode-parent tnode)))
      (cons nil nil)
      (let ((region (Insert-Region
                     (Create-Region-From-String "$\\ldots$")
                     :no-copy)))
```

```
        (cons (region-first-item region) (region-last-item region)))))

(defun self-handler (tnode)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let ((region
             (Insert-Region (Create-Region-From-String
                              (string-for-tnode tnode))
                            :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun string-handler (tnode string)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let ((region
             (Insert-Region (Create-Region-From-String
                              string)
                            :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun startprog-handler (tnode)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let ((region
             (Insert-Region (Create-Region-From-String
                              (format nil "\\obeylines~%"))
                            :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun endprog-handler (tnode)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let ((region
             (Insert-Region (Create-Region-From-String
                              (format nil "~%\\end~%"))
                            :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))
```

```
(defun emstring-handler (tnode string)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let ((region
              (Insert-Region (Create-Region-From-String
                                (string-concat "{\\bf " string "}"))
                             :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun in-handler (tnode depth)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
       (let ((region (Insert-Region
                       (Create-Region-From-String
                         (indent-n depth))
                     :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun sp-handler (tnode n)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
       (let ((region (Insert-Region
                        (Create-Region-From-String
                           (make-string n :initial-element #\space))
                        :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun cr-handler (tnode)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let ((region (Insert-Region
                      (Create-Region-From-String
                         (string-concat
                           (string #\newline)
                           (indent-n
                             (or (getvar 'compute-indent-level tnode) 0))))
```

```
                    :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun comment-handler (tnode)
  (if (tnode-is-elided? tnode)
      (cons nil nil)
      (let ((region
              (Insert-Region (Create-Region-From-String
                               (string-for-tnode
                                 (ppp::pprint-info-comment
                                   (otree:tnode-pp-info tnode))))
                             :no-copy)))
        (cons (region-first-item region) (region-last-item region)))))

(defun indent-n (n)
  (let ((string ""))
    (dotimes (i n)
      (setf string (string-concat string "\\quad ")))
    string))

(defun tnode-is-elided? (tnode)
  (cond
    ((otree:tnode-is-abstract-root? tnode)
     (unless (otree:tnode-is-error-NT? tnode)
       (getvar 'elide-node tnode)))
    (t (or (unless (otree:tnode-is-error-NT? tnode)
             (getvar 'elide-node tnode))
           (tnode-is-elided? (otree:tnode-parent tnode))))))

(add-back-end-function tex #'self-handler :self)
(add-back-end-function tex #'in-handler :in)
(add-back-end-function tex #'sp-handler :sp)
(add-back-end-function tex #'cr-handler :cr)
(add-back-end-function tex #'comment-handler :comment)
(add-back-end-function tex #'string-handler :string)
(add-back-end-function tex #'emstring-handler :emstring)
```

```
(add-back-end-function tex #'startprog-handler :startprog)
(add-back-end-function tex #'endprog-handler :endprog)
(add-back-end-function tex #'elide-handler :elide)
```

D Asple in *Ladle*

```
/* Copyright 1987-1990 by the Regents of the University of California.
 * All rights reserved.
 *
 *  PIPER:  Language Description
 *
 * Ladle definition for asple
 *
 */

LANGUAGE asple


LEXICAL

space = { \t\n\^L} => IGNORE ;

comment = "/*" ~ "*/" => SCREEN ;

integer = {0-9}+ ;

id = {a-zA-Z}{a-zA-Z0-9}* ;

keyBEGIN  = 'BEGIN'  IN id ;
keyBOOL   = 'BOOL'   IN id ;
keyDO     = 'DO'     IN id ;
keyELSE   = 'ELSE'   IN id ;
keyEND    = 'END'    IN id ;
keyFALSE  = 'FALSE'  IN id ;
keyFI     = 'FI'     IN id ;
keyIF     = 'IF'     IN id ;
```

```
keyINPUT  = 'INPUT'  IN id ;
keyINT    = 'INT'    IN id ;
keyOUTPUT = 'OUTPUT' IN id ;
keyREF    = 'REF'    IN id ;
keyWHILE  = 'WHILE'  IN id ;
keyTHEN   = 'THEN'   IN id ;
keyTRUE   = 'TRUE'   IN id ;


ABSTRACT

program = 'BEGIN' decls ";" stmts 'END' => program
        ;

decls = declaration + ";" => decls
      ;

stmts = statement + ";" => stmts
      ;

declaration = mode idlist => declaration
            ;

mode = 'BOOL' => ANNOTATE/*IMPLICIT*/
     | 'INT' => ANNOTATE/*IMPLICIT*/
     | 'REF' mode => ref
     ;

idlist = id + "," => idlist
       ;

statement = id ":=" expression => assignment
          | 'IF' expression 'THEN' stmts 'FI' => if_then
          | 'IF' expression 'THEN' stmts 'ELSE' stmts 'FI' => if_else
          | 'WHILE' expression 'DO' stmts 'END' => loop
          | 'INPUT' id => input
          | 'OUTPUT' expression => output
```

```
          ;

expression = constant => ANNOTATE/*IMPLICIT*/
           | id => ANNOTATE/*IMPLICIT*/
           | expression "+" expression => plus
           | expression "*" expression => times
           | "(" expression ")" => ANNOTATE
           | "(" expression "=" expression ")"  => equals
           | "(" expression "!=" expression ")" => not_equals
           ;

constant = 'TRUE' => ANNOTATE/*IMPLICIT*/
         | 'FALSE' => ANNOTATE/*IMPLICIT*/
         | integer => ANNOTATE/*IMPLICIT*/
         ;


CONCRETE

statement = transput
          | ifthen
          | ifthenelse
          | assignment
          | loop
          ;

transput = 'INPUT' id
         | 'OUTPUT' expression
         ;

ifthen  = 'IF' expression 'THEN' stmts 'FI'
            ;

ifthenelse  = 'IF' expression 'THEN' stmts  'ELSE' stmts 'FI'
           ;
```

```
assignment = id ":=" expression
           ;

loop = 'WHILE' expression 'DO' stmts 'END'
     ;

expression = top_expression
   ;

top_expression = factor
           | expression "+" factor
           ;

factor = primary
       | factor "*" primary
       ;

primary = id
        | constant
        | "(" expression ")"
        |  comparison
        ;

comparison = "(" expression "=" expression ")"
           | "(" expression "!=" expression ")"
           ;
```